Ch 03 신경망(Neural Network)

이전까지 퍼셉트론을 이용하여 함수를 표현하는 방식을 배웠습니다. 이런 방식에 문제점이 하나 있는데, 함수를 설계할 때의 가중치는 여전히 사람이 수동으로 설정해줘야 한다는 것입니다. 신경망(Nueral Network)는 가중치 매개변수의 적절한 값을 데이터로부터 자동으로 학습하여 이 문제를 해결해줍니다. 우선은 신경망의 개요와 신경망이 입력 데이터를 식별하는 처리 과정에 대해 정리해봅시다.

3.1 퍼셉트론에서 신경망으로

3.1.1 신경망의 예

입력층

은닉층

출력층

[그림 3-1] 신경망의 예

[그림 3-1]을 보면 다층 퍼셉트론과 크게 달라 보이지 않습니다. 신경망에서는 신호를 어떻게 전달할까요?

3.1.2 퍼셉트론 복습

[식 3.1]

[식 3.1]은 퍼셉트론은 x1, x2, 1 3개의 입력 신호를 받아서 가중치인 w1, w2, b를 곱합니다. 그 결과가 0보다 작거나 같으면 0, 크면 1을 출력합니다. (뉴런이 활성화됩니다.)

[식 3.2]

[식 3.3]

[식 3.2]는 [식 3.1]의 조건 분기 동작(0을 넘으면 1출력 그렇지 않으면 0출력)을 하나의 함수 h(x)로 나타낸 식입니다.

3.1.3 활성화 함수의 등장

위의 h(x)와 같이 입력 신호의 총합을 출력 신호로 변환하는 함수를 활성화 함수(activation function)이라고 합니다. 위의 [식 3.2]는 2단계로 처리됩니다. (1단계) 입력 신호와 가중치를 곱해 총합을 구하고, (2단계) 그 합을 활성화 함수에 입력해 결과를 냅니다. 다음 그림과 같은 흐름을 가집니다.

[그림 3-2] 활성화 함수의 처리과정

b

w1

h(x)

w2

3.2 활성화 함수

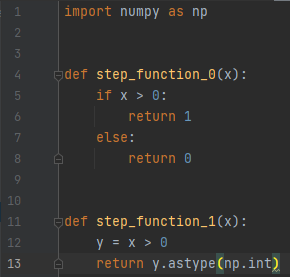
[식 3.3]은 임계값을 경계로 출력이 바뀌는데, 이를 계단 함수(step function)이라 합니다. 계단 함수가 아닌 다른 함수를 활성화 함수로 사용하는 것이 신경망을 만드는 열쇠라고 합니다!

3.2.1 시그모이드 함수

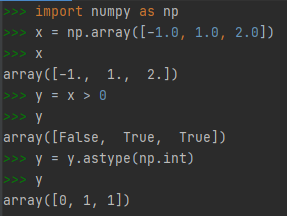
시그모이드 함수(sigmoid function)는 다음과 같습니다.

exp(-x)는 자연상수 입니다. 파이썬 코드로 계단 함수와 시그모이드 함수를 구현해보며 차이점을 알아보겠습니다.

3.2.2 계단 함수 구현하기

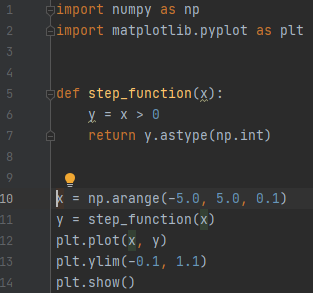


step\_function\_0는 넘파이 배열을 인수로 받을 수 없는 간단한 함수입니다. step\_function\_1은 밑의 인터프리터에서 보여주는 것처럼 넘파이 배열을 조건문을 통해 bool 값으로 바꿔주고, 그 bool 값으로 int형으로 형변환시키는 방법을 사용하여 계단 함수를 구현했습니다. (bool 값을 int로 변환시키면 True는 1로 False는 0으로 변환됩니다.)

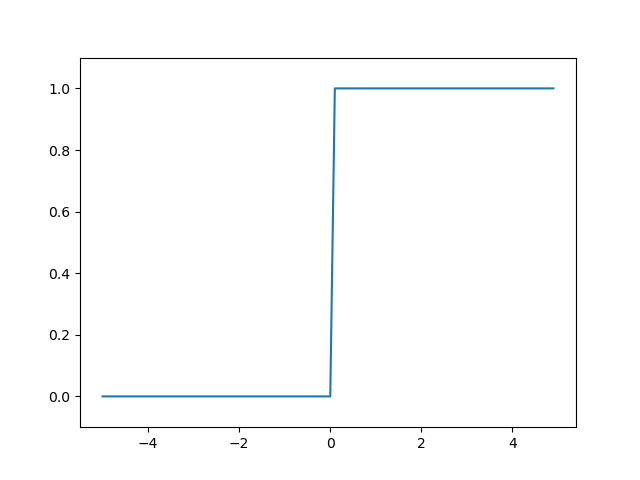


3.2.3 계단 함수의 그래프

matplotlib를 통해 계단 함수를 그려보겠습니다. 코드는 다음과 같습니다.



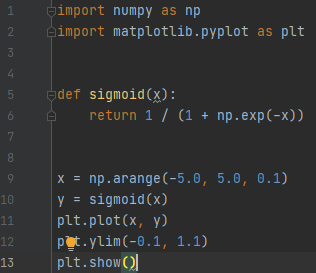
[그림 3-3] 계단 함수의 그래프



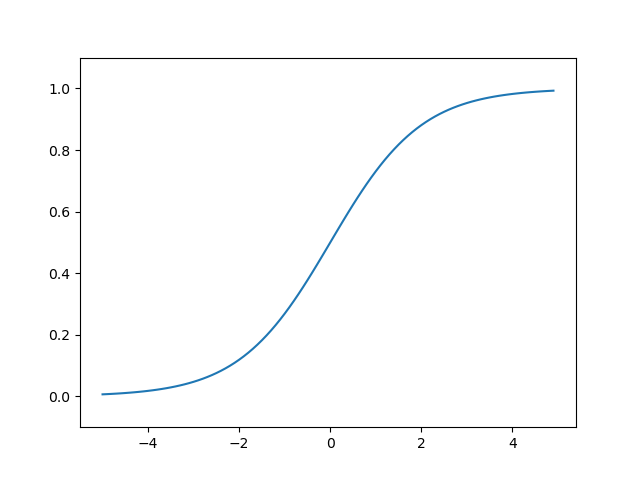
[그림 3-3]에서 볼 수 있듯이 계단함수는 0을 경계로 출력이 0에서 1로 바뀝니다.

3.2.4 시그모이드 함수 구현하기

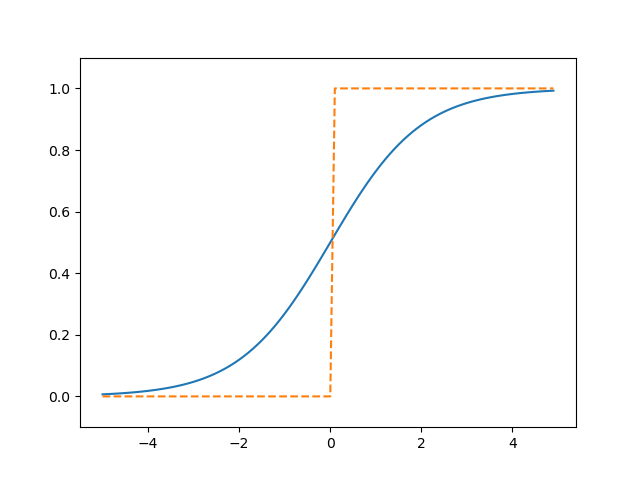
시그모이드 함수를 구현해보겠습니다. 파이썬으로 다음과 같이 간단히 구현할 수 있습니다.



[그림 3-4] 시그모이드 함수의 그래프



3.2.5 시그모이드 함수와 계단 함수 비교



둘의 가장 큰 차이점은 값의 연속성입니다. 시그모이드 함수는 출력이 연속적으로 변화하는 반면 계단 함수는 0에서 출력이 갑자기 바뀝니다. 신경망 학습에서 시그모이드 함수의 연속성은 아주 중요한 역할을 합니다.

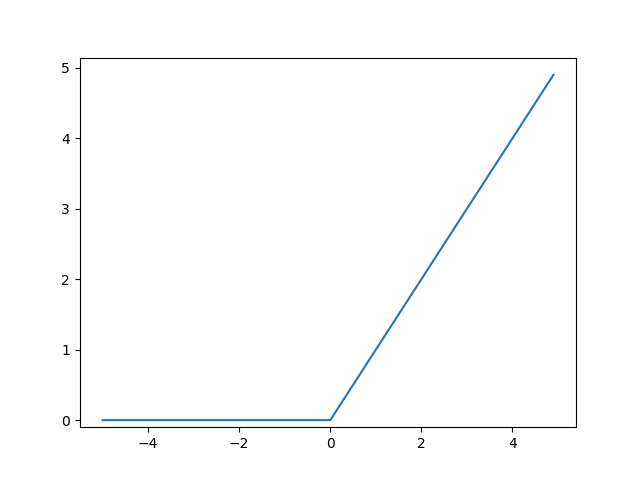
그래프를 보면 알 수 있는 한가지 공통점은 두 함수 모두 출력값이 0~1 사이인 것입니다. 그리고 두 함수 모두 입력이 작을 때의 출력은 0에 가깝거나 0이고, 입력이 커지면 출력이 1에 가까워지거나 1인 구조입니다.

3.2.6 비선형 함수

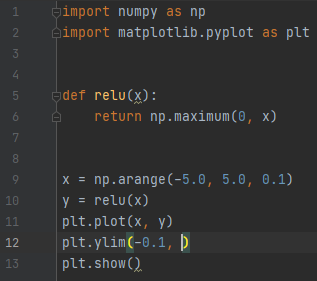
시그모이드 함수와 계단 함수의 공통점은 다른 것도 있습니다. 둘 다 비선형 함수라는 것입니다. 신경망에서 비선형 함수를 활성화 함수로 사용해야 하는데 그 이유는 선형 함수를 아무리 깊게 쌓아도 여러 층으로 구성하는 이유가 없기 때문입니다. h(x) = cx 라는 활성화 함수가 있다고 했을 때, 이를 통해 3층 네트워크를 쌓으면 y(x) = h(h(h(x))) 가 되는데, 이는 가 되지만 a=라 했을 때 y(x) = ax와 똑 같은 식이 됩니다. 이렇기 때문에 신경망에서 여러 층을 쌓을 때는 비선형 함수를 사용해야 합니다.

3.2.7 ReLU 함수

시그모이드는 오래전부터 이용되던 활성화 함수지만, 최근에는 ReLU(Rectified Linear Unit, 렐루)함수를 주로 이용합니다. ReLU 함수는 입력값이 0보다 크면 입력값을 그대로 출력하고 0보다 작으면 0을 출력하는 함수입니다. 다음 식처럼 나타낼 수 있고 그래프는 아래처럼 나옵니다.



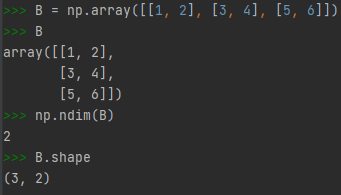
코드는 다음과 같습니다.



3.3 다차원 배열의 계산

넘파이를 이용하는 이유 중에 하나는 다차원 배열의 계산이 빠르다는 것입니다. 다차원 배열 계산을 숙달하면 신경망을 효율적으로 구축할 수 있다고 합니다.

3.3.1 다차원 배열



인터프리터로 간단하게 작성한 B는 ‘3x2 배열’인 2차원 배열입니다. 처음 차원에는 3개의 원소, 다음 차원에는 2개의 원소가 있다는 뜻입니다. 2차원 배열은 행렬(matrix)라 부르고 가로 방향을 행(row), 세로 방향을 열(column)이라 합니다.

행(row)

열(column)

3.3.2 행렬의 곱

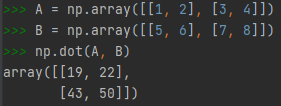
(1 x 5) + (2 x 7)

2x2행렬은 다음과 같이 계산합니다.

((앞 행렬의 첫 행 ) X (뒤 행렬의 첫 열) (앞 행렬의 첫 행 ) X (뒤 행렬의 두번째 열)

(앞 행렬의 두번째 행 ) X (뒤 행렬의 첫 열) (앞 행렬의 두번째 행 ) X (뒤 행렬의 두번째 열))

각 곱해진 원소의 합을 계산해 최종값이 나옵니다. 파이썬에서는 넘파이 함수를 사용해 구합니다.



행렬의 곱을 할 때 항상 주의해야 할 점은 행렬의 차원의 원소 수를 맞춰줘야 한다는 것입니다. 3x2행렬과 2x3행렬을 곱할 때 순서에 따라 출력되는 행렬의 차원의 원소수가 달라지는 것도 주의해야 합니다.

3.3.3 신경망에서의 행렬 곱

3

6

4

5

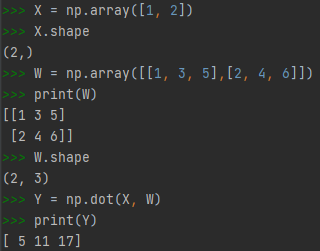
2

1

위 그림과 같은 신경망을 행렬의 곱으로 표현하면 밑처럼 나타낼 수 있습니다.

X W = Y

2 2x3 3



인터프리터로 간단하게 행렬 곱의 계산 예를 해볼 수 있습니다.

3.4 3층 신경망 구현하기

입력층 : 2개, 첫 번째 은닉층 : 3개, 두 번째 은닉층 : 2, 출력층 : 2 인 3층 신경망을 구현해봅시다.

import numpy as np  
from sigmoid\_function import sigmoid  
  
  
def identity\_function(x):  
 return x  
  
  
def init\_network():  
 network = {} # 신경망에서 쓰일 매개변수들의 dictionary를 만들어줍니다. 가중치는 W1, W2, W3, 편향은 b1, b2, b3  
 network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]]) # W1 key에 np.array value 입력  
 network['b1'] = np.array([0.1, 0.2, 0.3])  
 network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])  
 network['b2'] = np.array([0.1, 0.2])  
 network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])  
 network['b3'] = np.array([0.1, 0.2])  
  
 return network  
  
  
def forward(network, x):  
 W1, W2, W3 = network['W1'], network['W2'], network['W3']  
 b1, b2, b3 = network['b1'], network['b2'], network['b3']  
  
 a1 = np.dot(x, W1) + b1 # 행렬 곱을 계산합니다.  
 z1 = sigmoid(a1) # sigmoid 함수를 활성화 함수로 사용합니다.  
 a2 = np.dot(z1, W2) + b2  
 z2 = sigmoid(a2)  
 a3 = np.dot(z2, W3) + b3  
 y = identity\_function(a3) # 출력층으로 항등 함수로 설계했습니다.  
  
 return y  
  
  
network = init\_network()  
x = np.array([1.0, 0.5])  
y = forward(network, x)  
print(y) # [0.31682708 0.69627909]

3.5 출력층 설계하기

어떤 문제냐에 따라 출력층의 설계는 달라집니다. 회귀(regression)에는 항등 함수, 분류(classification)에는 소프트맥스 함수를 사용합니다. 분류는 데이터가 어느 클래스에 속하느냐는 문제이고, 회귀는 입력데이터에서 수치를 예측하는 문제입니다.

3.5.1 항등 함수와 소프트맥스 함수 구현하기

항등 함수(identity function)은 입력을 그대로 출력합니다. 소프트맥스 함수(softmax function)는 분류에서 사용하고 식은 다음과 같습니다.

소프트맥스 함수의 분자는 입력 신호 의 지수 함수, 분모는 모든 입력 신호의 지수 함수의 합입니다. 다음은 파이썬 코드로 구현해보겠습니다.

import numpy as np  
  
  
def softmax(a):  
 exp\_a = np.exp(a)  
 sum\_exp\_a = np.sum(exp\_a)  
 y = exp\_a / sum\_exp\_a  
  
 return y

간단하게 구현할 수 있지만 문제점이 있습니다.

3.5.2 소프트맥스 함수 구현 시 주의점

소프트맥스 함수를 구현할 때 지수 함수를 사용하게 되는데 아주 큰 값을 가지게 될 때가 있습니다. 이를 오버플로우(overflow) 오류가 생겼다고 합니다. 다음은 소프트맥스 함수를 개선한 수식입니다. 이때 C’는 입력 신호 중 최댓값을 이용하는 것이 일반적입니다.

수정 후 코드는 다음과 같습니다.

import numpy as np  
  
  
def softmax(a):

c = np.max(a)  
 exp\_a = np.exp(a - c)  
 sum\_exp\_a = np.sum(exp\_a)  
 y = exp\_a / sum\_exp\_a  
  
 return y

3.5.3 소프트맥스 함수의 특징

소프트맥스 함수의 출력은 0에서 1.0 사이의 실수이고, 모든 출력의 총합은 1입니다. 이 성질로 인해 함수의 출력을 ‘확률’로 해석할 수 있습니다. 소프트맥스 함수는 적용해도 원소의 대소관계는 바뀌지 않습니다. 출력층을 거쳐도 대수 관계가 바뀌지 않으므로 추론(inference)단계에서 시간이 오래 걸리는 지수 계산을 포함한 소프트맥스 함수를 생략하기도 합니다.

3.5.4 출력층의 뉴런 수 정하기

출력층의 뉴런 수는 풀고자 하는 문제에 맞춰 적절히 설정해야 합니다. 이후에 예로 사용될 손 글씨 숫자 인식의 경우에는 0~9 까지의 숫자를 구분(분류)하는 문제이기 때문에 출력층의 뉴런 수를 10개로 설정해야 합니다.

3.6 손글씨 숫자 인식

손글씨 숫자 분류를 통해 신경망을 구현해보도록 합시다. 우선은 추론 과정만 구현하는데 이 추론 과정을 순전파(forward propagation)이라고 합니다.

3.6.1 MNIST 데이터셋

MNIST는 손글씨 숫자 이미지 집합입니다. 60000개의 훈련 이미지, 10000개의 시험 이미지로 이루어져 있습니다. 파이썬에는 피클(pickle)이라는 기능이 있습니다. 프로그램 실행 중 특정 객체를 파일로 저장하는 기능으로 이번 예에서 저장해둔 pickle 파일을 로드해 실행 당신의 객체를 복원해 사용할 것입니다. 이 부분의 데이터셋 관련 코드는 넘기겠습니다.

3.6.2 신경망의 추론 처리

from dataset.mnist import \*  
from sigmoid\_function import sigmoid  
from softmax\_function import softmax  
import numpy as np  
  
  
def get\_data():  
 (x\_train, t\_train), (x\_test, t\_test) = \  
 load\_mnist(normalize=True, flatten=True, one\_hot\_label=False)  
 return x\_test, t\_test  
  
  
def init\_network():  
 with open("sample\_weight.pkl", 'rb') as f: # 미리 저장되어있는 가중치 데이터  
 network = pickle.load(f)  
  
 return network  
  
  
def predict(network, x):  
 W1, W2, W3 = network['W1'], network['W2'], network['W3']  
 b1, b2, b3 = network['b1'], network['b2'], network['b3']  
  
 a1 = np.dot(x, W1) + b1  
 z1 = sigmoid(a1)  
 a2 = np.dot(z1, W2) + b2  
 z2 = sigmoid(a2)  
 a3 = np.dot(z2, W3) + b3  
 y = softmax(a3)  
  
 return y  
  
  
x, t = get\_data()  
network = init\_network()  
  
accuracy\_cnt = 0  
for i in range(len(x)):  
 y = predict(network, x[i])  
 p = np.argmax(y) # 확률이 가장 높은 원소의 인덱스를 얻는다  
 if p == t[i]:  
 accuracy\_cnt += 1  
  
print("Accuracy : " + str(float(accuracy\_cnt) / len(x)))



정규화(normalization)는 전처리(pre-processing) 방식 중 하나로 데이터를 특정 범위로 변환하는 처리를 말합니다. 이번에는 0~255의 데이터를 0~1.0의 범위로 변환했습니다. 다음으로 pickle 파일에 저장되어 있던 매개변수 데이터를 가져와 정확도를 측정합니다. 출력에서 확률이 가장 높은 원소를 추출해 시험 데이터와 비교를 해보는 것으로 신경망의 정확도를 측정할 수 있습니다.

3.6.3 배치 처리

이미지 데이터 1장의 신경망의 처리 흐름은 다음과 같습니다.

784 784 X 50 50 X 100 100 X 10 -> 10

X W1 W2 W3 -> Y

MNIST 데이터 하나(1장)의 크기는 28x28로 평탄화(flatten)시키게 되면 원소 784개로 구성된 1차원 배열입니다. 그렇다면 100장 분량의 데이터는 어떤 식으로 처리될까요?

100 X 784 784 X 50 50 X 100 100 X 10 -> 100 x 10

X W1 W2 W3 -> Y

입력데이터가 100x784로 바뀌고 출력 데이터가 100x10 이 됩니다. 이는 100장의 입력데이터에 대한 결과가 한 번에 출력됨을 뜻합니다. 이처럼 하나로 묶인 입력 데이터를 배치(batch)라고 합니다. 입력데이터의 크기를 키움으로써 계산 횟수를 줄일 수 있어 큰 이점을 얻을 수 있습니다. 배치 처리를 구현한 코드입니다.

x, t = get\_data()  
network = init\_network()

batch\_size = 100  
accuracy\_cnt = 0

for i in range(0, len(x), batch\_size):

x\_batch = x[i:i+batch\_size]

y\_batch = predict(network, x\_batch)  
 p = np.argmax(y\_batch, axis=1) # 각 행에서 최댓값의 index

accuracy\_cnt += np.sum(p == t[i:i+batch\_size])  
  
print("Accuracy : " + str(float(accuracy\_cnt) / len(x)))

이번 절에서는 퍼셉트론과 신경망과의 차이점을 알아보고 활성화 함수를 통해 신경망의 추론 과정에 대해 공부했습니다. 그리고 입력 데이터와 출력 데이터에 따라 어떻게 신경망을 구축해야 하는지도 맛봤습니다.